# Autonomic Management of Reconfigurable Embedded Systems using Discrete Control: Application to FPGA

Xin An, Eric Rutten, Jean-Philippe Diguet, Nicolas Le Griguer, Abdoulaye Gamatié


# Autonomic Management of Reconfigurable Embedded Systems using Discrete Control : Application to FPGA

**Xin An, Eric Rutten, Jean-Philippe Diguet, Nicolas le Griguer, Abdoulaye Gamatié**

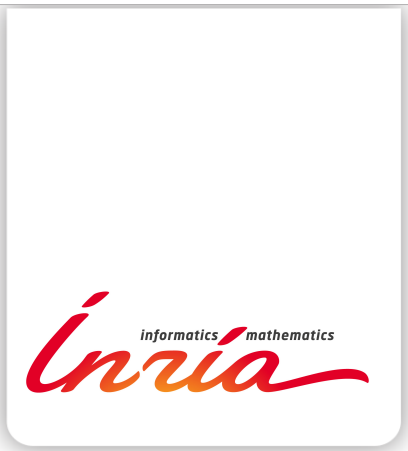
**RESEARCH REPORT**

**N° 8308**

May 2013

Project-Teams Ctrl-A, Lab-STICC and LIRMM

ISSN 0249-6399 ISRN INRIA/RR--8308--FR+ENG




# Autonomic Management of Reconfigurable Embedded Systems using Discrete Control : Application to FPGA

Xin An, Eric Rutten, Jean-Philippe Diguet, Nicolas le Griguer, Abdoulaye Gamatié

Project-Teams Ctrl-A, Lab-STICC and LIRMM

Research Report n° 8308 — May 2013 — 24 pages

**Abstract:** This paper targets the autonomic management of dynamically partially reconfigurable hardware architectures based on FPGAs. Such hardware-level autonomic computing has been less often studied than at software-level. We consider control techniques to model the considered behaviours of the computing system and derive a controller for the control objective enforcement. Discrete Control modelled with Labelled Transition Systems is employed in this paper. Such models are amenable to Discrete Controller Synthesis algorithms that can automatically generate a controller enforcing the correct behaviours of a controlled system. A general modelling framework is proposed for the control of FPGA based computing systems. We consider system application described as task graphs and FPGA as a set of reconfigurable areas that can be dynamically partially reconfigured to execute tasks. We encode the computation of an autonomic manager as a DCS problem w.r.t. multiple constraints and objectives e.g., mutual exclusion of resource uses, power cost minimization. We validate our models and manager computations by using the BZR language and an experimental demonstrator implemented on a Xilinx FPGA platform.

**Key-words:** Hardware Architectures, Dynamically Partially Reconfigurable FPGA, Discrete Control

**RESEARCH CENTRE GRENOBLE – RHÔNE-ALPES**

Inovallée
655 avenue de l'Europe Montbonnot
38334 Saint Ismier Cedex

# Gestion Autonomique des Systèmes Embarqués Reconfigurables utilisant le Contrôle Discret : Application aux FPGA

**Résumé :** Nous traitons de la gestion autonomique des architectures matérielles dynamiquement et partiellement reconfigurables à base de FPGAs. Cette forme d'informatique autonomique au niveau matériel a été peu étudiée comparé au niveau logiciel. Nous considérons des techniques de contrôle pour modéliser les comportements du système de calcul et pour dériver un contrôleur pour le maintien de l'objectif de contrôle. Nous utilisons des techniques de contrôle discret modélisé avec des systèmes de transitions étiquetées. Ces modèles se prêtent à une algorithmique de synthèse de contrôleurs discrets (SCD) qui peut générer automatiquement un contrôleur qui force les comportements corrects d'un système contrôlé. Un cadre général de modélisation est proposé pour le contrôle des systèmes informatiques à base de FPGA. Nous considérons que l'application est décrite par un graphe de tâches, et le FPGA comme un ensemble de zones reconfigurables, qui peuvent être dynamiquement et partiellement reconfigurées pour exécuter des tâches. Nous formulons le calcul d'un gestionnaire autonomique comme un problème de SCD concernant des contraintes et objectifs multiples, par exemple, l'exclusion mutuelle de l'utilisation des ressources, la minimisation du coût en énergie. Nous validons nos modèles et les calculs du gestionnaire en utilisant le langage BZR et un démonstrateur expérimental mis en œuvre sur une plate-forme FPGA Xilinx.

**Mots-clés :** Architectures matérielles, FPGA reconfigurable dynamiquement et partiellement, contrôle discret

# Contents



# 1 Control of autonomic hardware

**Controlling FPGAs.** We apply the autonomic framework to the context of FPGAs (Field Programmable Gate Arrays), hardware devices that compute a logic function by configuring its gates in a programmable way. A recent progress is *dynamically partially reconfigurable* (DPR) FPGAs. They support partial reconfigurations where only part of gates are reconfigured and reconfigurations to be performed at runtime. Autonomic computing has been seldom applied to such hardware systems, though they represent a significant case of its relevance.

**Control for autonomic management.** We adopt control techniques to design the *MAPE-K* (Monitor, Analyse, Plan, Execute, based on Knowledge). Formal models are used to describe the possible behaviours of the system under design, and control objectives giving the adaptation policy are specified separately. A controller is then derived based on the system models and objectives. The use of classical control techniques and models, typically these based on continuous time dynamics and differential equations, has been explored for various computing systems [8] and sometimes applied for hardware architectures [6]. A similar approach can be adopted by using *discrete control* techniques, where systems are considered from the viewpoint of events and states. The behaviours can then be modelled in the form of Petri nets or automata for, typically, synchronisation or coordination [17].

**Discrete control for autonomic FPGAs.** In this paper, we apply *discrete control* for the autonomic management of DPR FPGA based embedded systems. A systematic modelling framework is proposed, where system application behaviour, task implementations and executions, architecture reconfigurations and environment are modelled separately by using *Labelled Transition Systems* (LTS) or *automata*. *Discrete Controller Synthesis* (DCS) supported by a programming language and synthesis tool has been applied to compute an autonomic manager.

A video processing system has been implemented on a Xilinx FPGA platform to validate our proposal. Some results in the research report have been published in [2].

Section 2 recalls the backgrounds on FPGA architectures, discrete control and its relation to MAPE-K. Section 3 presents our modelling and autonomic manager computing framework through an illustrative example. Section 4.1 describes a real-life case study. Section 5 discusses related work, and Section 6 concludes.

# 2 Background notions

## 2.1 FPGA-based architectures

**Basic reconfigurable cell.** A FPGA is composed of an array of logic cells and programmable routing channels to implement custom hardware functionalities. The basic components of a logic cell are the LUT: a memory used as a programmable device to implement any logic function between inputs and outputs of a cell, and the D flip-flop: to hold a state between two clock cycles.

A program consists of one or more *bitstream*s, which are binary files storing information to configure the LUTs and the routing switches. The bitstreams are generated by design tools such as the Xilinx Embedded Development Kit (EDK), which includes a tool suite called Xilinx Platform Studio (XPS) used to design an embedded system. Recent large FPGAs contain more than 200K logic cells that can be combined and interconnected to implement very complex designs. Multi-core architectures with tens of large hardware accelerators and processors can be implemented.

**Run-time partial reconfiguration.** In the new generation of FPGAs, the hardware configuration can be updated at run-time by using the partial reconfiguration feature. A portion or region of the FPGA which implements some logic functions can be swapped with another one. If multiple functions are called sequentially, the same region can be reused so that the required size can be minimised. The best advantage of this type of reconfiguration is its ability to reconfigure hardware during the running of the static part, i.e., the part which does not contain any reconfigurable area. It assumes that the hardware reconfiguration does not disturb the execution of the application. The bitstreams therefore cover only some regions of the FPGA array.

Such DPR FPGAs make them suitable for addressing constraints on resources (re-using some areas for different functions for applications that can be partitioned into phases) by adapting resources to available parallelism according to environment variations. DPR FPGAs represent a trade-off in that they are slower than dedicated Application-Specific Integrated Circuit (ASIC) hardware, but much faster than software running on general purpose CPUs.

**Management of reconfiguration.** From a technical viewpoint, each hardware configuration file used for the different implementations of the partially reconfigurable regions is stored into a compact flash card. It can be loaded with a processor (e.g. microblaze, which is a 32-bit soft-core processor as implementable on Xilinx FPGAs). It performs the reconfiguration using the ICAP (Internal Configuration Access Port) as in Figure 1.

The runtime management of reconfiguration involves a control loop, taking decision according to events monitored on the architecture, choosing the appropriate next configuration to install, and executing appropriate reconfiguration actions. The architecture dynamism increases the design complexity, for which a complete tool-chain is lacking [14]. Due to the relative novelty of DPR technologies, the management of reconfiguration has to be designed manually for important parts.

Amongst different approaches to address this issue, we investigate the adoption of an autonomic computing approach for the design of reconfiguration control. The MAPE-K structure is

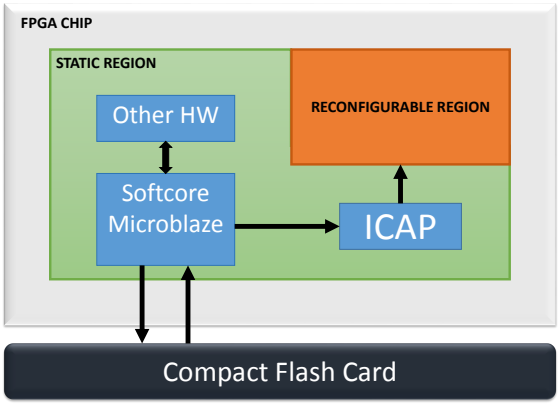



Figure 1: FPGA with a microblaze softcore.

based on behavioural models (in the form of automata) for the knowledge about the reconfigurability of these hardware platforms, and discrete control techniques for designing the adaptation policies.

## 2.2 Discrete control

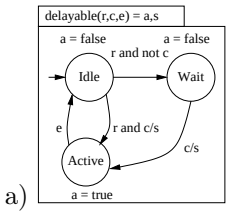
**Automata and data-flow nodes.** We first briefly introduce the basics of the automata-based modelling framework using formalisms from [1]. Behaviours are modelled in terms of Finite State Machines (FSM), or more precisely Labelled Transition Systems (LTS). They are defined by a finite set of states, between which there are transitions (from source state to target state) with a label of the form `c / a`: a firing condition `c` and an action `a`. When the FSM is in some current state, if there is a transition for which the condition is true, then it is taken and the next current state will be the target state. At the same time the action part will take the value `true`.

Figure 2(a) illustrates this in the case of the control behaviour of a delayable task. It describes the control of a `delayable` task, which can either be idle, waiting or active. When it is in the initial Idle state, the occurrence of the `true` value on input `r` *requests* the starting of the task. Another input `c` can either allow the activation, or temporarily block the request and make the automaton go to a waiting state. Input `e` notifies termination. The outputs represent, resp., `a`: activity of the task, and `s`: triggering starting operation in the system's API.

As a concrete specification tool, we use the Heptagon programming language [4]. It supports the programming of mixed synchronous data-flow equations and automatawith parallel and hierarchical composition. The basic behaviour is that at each reaction step, values in the input flows are used, as well as local and memory values, in order to compute the values in the output flows for that step. Inside the nodes, this is expressed as a set of equations defining, for each output and local, the value of the flow, in terms of an expression on other flows, possibly using local flows and state values from past steps.

Figure 2(a) shows a small program in this language, encoding exactly the FSM of Figure 2(a).

Such automata and data-flow reactive nodes can be reused by instantiation, and composed in parallel (noted formally "|", and in the concrete syntax `";"`) and in a hierarchical way, as illustrated in the body of the node in Figure 2(c), with two instances of the `delayable` node. They run in parallel: one global step corresponds to one local step for every node. The compilation produces executable code in target languages such as C or Java, in the form of an initialisa-




a)

```
node delayable(r,c,e:bool) returns (a,s:bool)
 let automaton
   state Idle    do a = false ; s = r and c
    until r and c then Active
     | r and not c then Wait
   state Wait    do a = false ; s = c
    until c then Active
   state Active  do a = true ; s=false

    until e then Idle
   end  tel
```

b)

```
twotasks(r1, e1, r2, e2)
                = a1, s1, a2, s2
enforce not (a1 and a2)
with c1, c2

(a1, s1) = delayable(r1, c1, e1) ;
(a2, s2) = delayable(r2, c2, e2)
```

c)

Figure 2: Delayable task: a) graphical / b) textual syntax; c) exclusion contract.

tion function *reset*, and a *step* function implementing the transition function of the resulting automaton. It takes incoming values of input flows gathered in the environment, computes the next state on internal variables, and returns values for the output flows. It is called at relevant instants from the infrastructure where the controller is used.

**Control and contracts.** The formalism of LTSs can be used to apply *discrete controller synthesis* (DCS), a formal operation on automata [3, 12]: given a FSM representing possible behaviours of a system, its variables are partitioned into controllable ones and uncontrollable ones. For a given control objective (e.g., staying invariably inside a subset of states, considered "good"), the DCS algorithm automatically computes, by exploration of the state graph, the constraint on controllable variables, depending on current state, for any value of the uncontrollables, so that remaining behaviours satisfy the objective. This constraint is inhibiting the minimum possible behaviours, therefore it is called *maximally permissive*. Algorithms are related to model checking techniques for state space exploration.

BZR (`http://bzr.inria.fr`) extends Heptagon with a new behavioural contract [4]: its compilation involves DCS. Concretely, the BZR language allows for the declaration, using the `with` statement, of controllable variables, the value of which are not defined by the programmer. These free variables can be used in the program to describe choices between several transitions. They are then defined, in the final executable program, by the controller computed by DCS, according to the expression given in the `enforce` statement. BZR compilation invokes a DCS tool, and inserts the synthesised controller in the generated executable code, which has the same structure as above: *reset* and *step* functions.

Figure 2(c) shows an example of contract coordinating two instances of the `delayable` node of Figure 2(a). The `twotasks` node has a `with` part declaring controllable variables $c_1$ and $c_2$, and the `enforce` part asserts the property to be enforced by DCS. Here, we want to ensure that the two tasks running in parallel will not be both active at the same time: `not` (A1 `and` A2). Thus, $c_1$ and $c_2$ will be used by the computed controller to block some requests, leading automata of tasks to the waiting state whenever the other task is active. The constraint produced by DCS can have several solutions: the BZR compiler generates deterministic executable code by giving priority, for each controllable variable, to value `true` over `false`, and between them, by following the order of declaration in the `with` statement.

## 2.3 Discrete control as MAPE-K

Figure 3(a) shows the MAPE-K architecture of an autonomic system with a loop defining basic notions of Managed Element (ME) and Autonomic Manager (AM). The managed element, system or resource is monitored through sensors. An analysis of this information is used, in combination with knowledge about the system, to plan and decide upon actions. These recon-

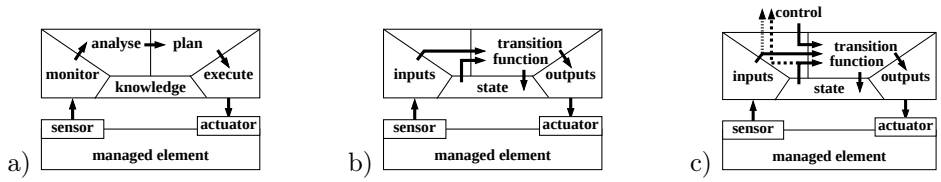



Figure 3: Autonomic system: a) the MAPE-K manager; b) FSM autonomic manager; c) controllable AM.

figuration operations are executed, using as actuators the administration functions offered by the system API. Self-management issues include self-configuration, self-optimisation, self-healing (fault tolerance and repair), and self-protection.

Autonomic managers work in closed loop: for this, one design methodology is to apply techniques from Control Theory [8], with the advantage of ensuring interesting properties on the resulting behaviour of the controlled system e.g., stability, convergence, reachability or avoidance of some evolutions. In most cases, continuous models are used, typically for quantitative aspects. More recently, some works relied on Discrete Event Systems (DES), using supervisory control [3], typically for logical or synchronisation purposes e.g., deadlock avoidance in multi-threaded programs [17]. They are based on reactive systems models such as Petri nets or Finite State Machines (FSM), which we also call automata. As shown in Figure 3(b), this instantiates the general autonomic loop with knowledge on possible behaviours represented as a formal state machine, and planning and execution in the form of the automaton transition function with a control output, which will trigger the actuator.

Basic features required for a system to be managed in an autonomic fashion have been identified in previous work e.g., in the context of component-based autonomic management [15]: for an ME to be manageable it must be observable and controllable. The manager transforms flows of observations into flows of control choices and actions. Observability translates into outputs, as shown by dashed arrows in Figure 3(c) for an FSM AM, exhibiting (some) of the knowledge and sensor information (raw, or analysed); this can feature state information on the AM itself or of MEs below. Controllability translates to having the AM accept some influence on the decision, and it corresponds to additional input for control, as in Figure 3 for an FSM AM. Its values can be used in the guards and exhibit choices between different transitions.

This builds up to a hierarchical framework as in the structure shown in Figure 4. Given that AMs have been made observable and controllable, an upper-level AM can perform their

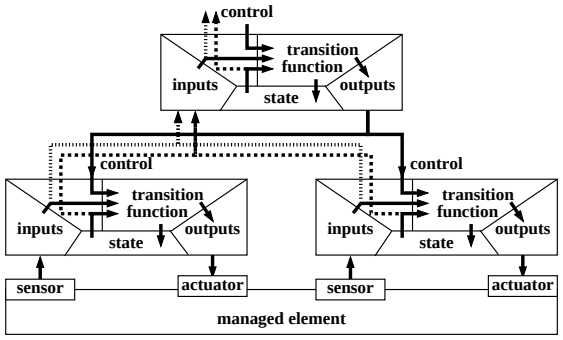



Figure 4: Autonomic coordination for multiple AMs.

coordination using their additional control input to enforce a policy. Considering the case of FSM managers makes it possible to encode the coordination problem as a DCS problem. The controller of this upper-level AM is synthesised by DCS.

# 3 DCS for managing DPR architectures

## 3.1 DPR FPGAs

We present informally the class of computing systems of interest through an illustrative example. They are inspired by the self-adaptive embedded systems in [6]. However, we address the problem in a different and original way.

**Hardware architecture.** We consider a multiprocessor architecture implemented on an FPGA (e.g. Xilinx Zynq device), which is composed of a general purpose processor $A0$ (e.g. ARM Cortex A9), and a reconfigurable area divided into four tiles: $A1$–$A4$ (see Figure 5). The communications between architecture components are achieved by a *Network-on-Chip* (NoC). Each processor and reconfigurable tile implements a NoC Interface (NI). A fixed dual port memory buffer is associated with each tile, which means that at most two tasks can simultaneously access data stored in the shared memory. Reconfigurable tiles can be combined and configured to implement and execute tasks by loading predefined bitstreams, such as tiles A1 and A2 of Figure 5.

The architecture is equipped with a battery supplying the platform with energy. Regarding power management, an unused reconfigurable tile $Ai$ can be put into sleep mode with a *clock gated mechanism* such that it consumes a minimum static power.

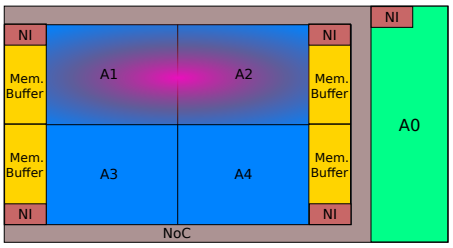



Figure 5: Architecture structure and execution example.

**Application software.** We consider system functionality described as a *directed, acyclic task graph* (DAG). A DAG consists of a set of *nodes* representing the set of tasks to be executed, and a set of directed *edges* representing the precedence constraints between tasks. Note that we do not restrict the abstraction level of tasks associated with the nodes, and a task can be an atomic operation, or a coarse fragment of system functionality. Figure 6 shows an example consisting of four tasks.

In our framework, we suppose each task performs its computation with the following four control points:

- *being requested* or invoked;
- *being delayed*: requested but not yet executed;
- *being executed*: to be executed on the architecture;

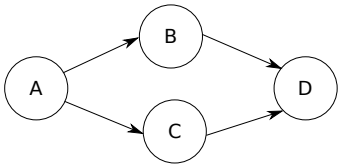



Figure 6: DAG application specification.

- *notifying execution finish*, once it reaches its end.

Occurrences of control points *being requested* and *notifying finishes* depend on runtime situations, and are thus unpredictable and uncontrollable. The way of *delaying* and *executing* tasks is taken charge by a runtime manager aiming to achieve system objectives.

**Task implementation.** Given a hardware architecture, a task can be implemented in various ways characterised by various parameters of interest, such as used reconfigurable tiles ($ur$), worst case execution time (WCET) ($wt$), and power peak $pp$. For example, two implementations of task $A$ can be:

- $A$ on $A1$: $wt = 50$, $pp = 20$;
- $A$ on $A3 + A4$: $wt = 10$, $pp = 30$;

In this preliminary work, we assume that WCET represents the time cost induced from the start of bitstream loading to the end of task execution. Among the possible task implementations, a runtime manager is in charge of choosing the best implementation at runtime according to system objectives.

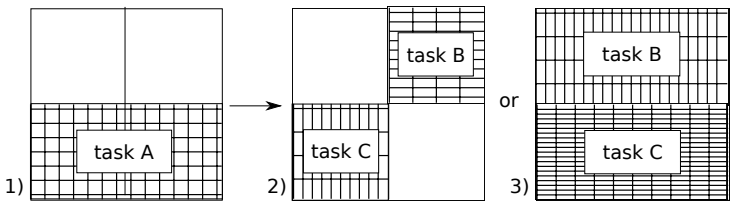



Figure 7: Configurations and reconfigurations.

**System reconfiguration.** Figure 7 shows three system configuration examples. In configuration 1, task A is running on tiles A3 and A4 while tiles A and B are set to the sleep mode. Configurations 2 and 3 show two scenarios with tasks B and C running in parallel. Once task A finishes its execution according to the graph of Figure 6, the system can go to either configuration 2 or configuration 3 depending on the system requirements. For example, if the current state of the battery level is low, the system would choose configuration 2 as configuration 3 requires the complete FPGA working surface and therefore consumes more power.

**System objectives.** System objectives define the system functional and non-functional requirements. This section gives the objectives considered in the paper, and categorises them as logical and optimal control objectives. Generally speaking, logical objectives concern state exclusions, whereas optimal objectives target the states associated with optimal costs.

Considered logical control objectives are as follows:

1. resource usage constraint: exclusive uses of reconfigurable areas $A1$-$A4$;
2. dual accesses to the shared memory (i.e., at most two functions running in parallel);
3. energy reduction constraint: switch areas to
   (a) sleep mode when executing no task;
   (b) active mode when needed;
4. reachability: system execution can always finish once started;
5. power peak constraint: power peak of hardware platform is constrained w.r.t battery levels;

Optimal control objectives of interest are as follows:

6. minimise power peak of hardware platform;
7. minimise WCET of system executions;
8. minimise worst case energy consumption of system executions.

## 3.2 System modelling as a DCS problem

We specify the modelling of the computing system behaviour and control in terms of labelled automata. System objectives are defined based on the models. We focus on the management of computations on the reconfigurable tiles and dedicate the processor area $A0$ exclusively to the resulting controller.

**Architecture behaviour.** The architecture (see Figure 5) consists of a processor $A0$, four reconfigurable tiles $\{A1, A2, A3, A4\}$ and a battery. Each tile has two execution modes, and the mode switches are controllable. Figure 8(a) gives the model of the behaviour of tile $Ai$. The mode switch action between Sleep (*Sle*) and Active (*Act*) depends on the value of the Boolean controllable variable $c\_a_i$. The output $act_i$ represents its current mode.

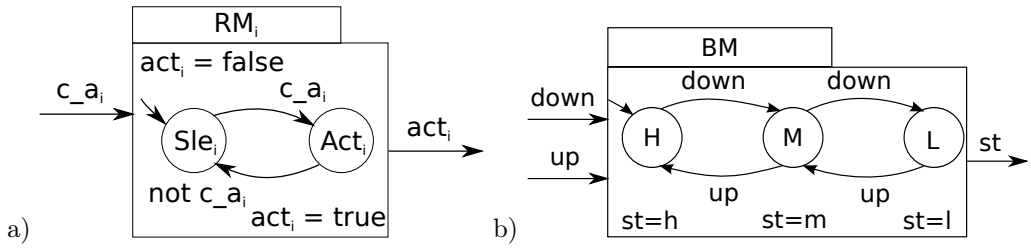



Figure 8: Models $RM_i$ for tile $Ai$, and $BM$ for battery.

The battery behaviour is captured by the automaton in Figure 8(b). It has three states labelled as follows: $H$ (high), $M$ (medium) and $L$ (low). The model takes input from the battery sensor, which emits level *up* and *down* events, and keeps track of the current battery level through output *st*.

**Application behaviour.** Software application is described as a DAG, which specifies the tasks to be executed and their execution sequences and parallelism. We capture its behaviour by defining a *scheduler automaton* representing all possible execution scenarios. It does so by keeping tracking of application execution states and emitting the *start* requests of tasks in reaction to the task *finish notifications*.

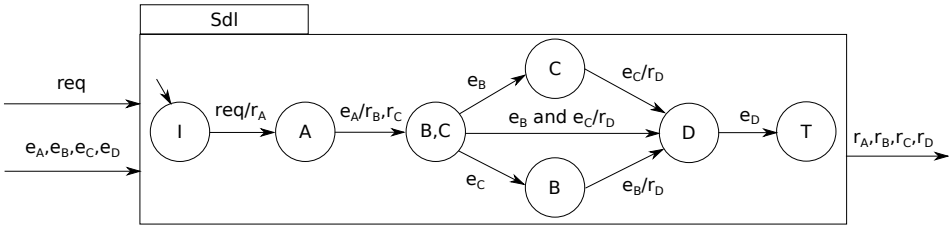



Figure 9: Scheduler automaton *Sdl* capturing application execution behaviours.

Figure 9 shows the scheduler automaton of the application in Figure 6. It starts the execution of the application by emitting event $r_A$, which requests the *start* of task $A$, upon the receipt of application request event $req$ in the idle state $I$. Upon the receipt of event $e_A$ notifying the *end* of $A$'s execution, events $r_B$ and $r_C$ are emitted together to request the execution of tasks $B$ and $C$ in parallel. Task $D$ is not requested until the execution of both $B$ and $C$ is finished, denoted by events $e_B$ and $e_C$. It reaches the final state $T$, implying the end of the application execution, upon the receipt of event $e_D$.

**Scheduler Automaton Derivation.** The scheduler automaton or LTS of a DAG described application captures the dynamic execution behavior of the application. Its states represent the tasks that are executing. They are denoted and labeled by the names of these tasks. It has an initial state $I$, i.e., the idle state, which means the application has not been invoked, and an end state $T$, which means that the application has finished its execution. The automaton input events are the task end events $e_i$ and the application request event $req$, while its output events are the task request events $r_i$. Its transitions are of the form $g/a$, where $g$ is a firing condition, and $a$ is an action. A firing condition is a boolean expression of input events, and an action is a conjunction of output events. *Note*: 1) we suppose the application is only invoked once. If it is allowed to be repeatedly invoked, the end state would be the same to the initial state. 2) if the graph has a task that has more than one instance, the instances are then seen as different tasks by the algorithm.

Algorithm 1 illustrates how to construct the scheduler automaton for a DAG. It derives the automaton from initial state $I$ to end state $T$ by exploring the state space of the application execution w.r.t. the DAG.

- *Inputs:* a directed, acyclic task graph $< T, C >$, where $T$ and $C$ represent respectively the set of tasks, the set of edges.
- *Local variables and functions* used in the algorithm: $s$: a state, with element *label* represents the tasks executing, and element *taskSet* to represent the set of tasks *associated* to

the state (i.e., executing in the state); *stateQueue*: a FIFO queue, keeping track of the states to be processed, with functions *popup*(), *add*($s$) to return and delete the first state element, and add state $s$ to the end of the queue; $readyTaskSet$: $t_i.prec$: the set of tasks immediately precedes $t_i$; traversed($s$): a function returns the states from I to $s$ (included), which represents the tasks that have finished and executing, with *taskSet* to return union of tasks associated with each state; drawTrans(source,sink,trans. label), drawState(I); drawnStates: the states that have been drawn out; toState(task set): to associate/label the task set with the state; $tc$: a set of tasks, or a task combination; powerSet($tc$): the power set of $tc$ - $\emptyset$.

At line 1, the initial state, i.e., idle state $I$ is drawn denoted by $drawn(I)$. The set of drawn states *drawnStates* is thus initialized to $\{I\}$ at line 2. State queue *stateQueue* stores the states that have been drawn but not processed. It is initialized to have element $I$ at line 3. Variable $readyTaskSet$ represents the set of tasks that are *enabled* to execute once some event happens. A task is *enabled* if all its precedent tasks have finished their executions. Lines 4 to 8 set $readyTaskSet$ to the set of tasks that have no precedent tasks, as such tasks can be executed immediately once the application is invoked/requested denoted by the receipt of event *req*. Lines 9 to 41 deal with the sequential processing of the states stored in *stateQueue*. The processing of a state concerns the drawing of its immediate following states and the transitions, and put the new drawn states in the queue. The automaton derivation finishes when the queue becomes empty.

In the following, we describe how state $s$ from queue *stateQueue* is processed. Line 10 evaluates the first state of the queue to $s$ and removes it from the queue. Three types of states are distinguished and processed accordingly. They are initial state $I$, end state $T$ and the rest. Lines 12 to 16 deal with the processing of idle state $I$. The algorithm firstly computes its following state *nextState* by evaluating the $readyTaskSet$ (got from Lines 4 to 8) to its *taskSet* at line 12. *nextState* represents the state once the application is invoked. Line 13 draws the state, and line 14 draws the transition from state $I$ to *nextState* with label $req/\{r_i|t_i \in readyTaskSet\}$, where $r_i$ is the request event of task $t_i$, denoted by $drawTrans(I, nextState, req/r_{readyTaskSet})$. Lines 15 and 16 add *nextState* to *drawnStates* and the end of *stateQueue*. If state $s$ is the end state (line 17), the algorithm simply proceeds.

Lines 20 to 39 deal with the processing of state $s$ that represents the application executing state between $I$ and $T$. In general, the algorithm explores all the possible subsets of the executing tasks in $s$ (denoted by $s.taskSet$), and computes the following states accordingly. $powerSet(s.taskSet)$ represents the power set of $s.taskSet$ without $\emptyset$. Given an element $tc$ which represent a subset of the executing tasks in $s$, lines 21 to 38 deal with the drawing of the following states of $s$ w.r.t. the simultaneous finishes of the executions of tasks in $tc$. Lines 21 to 26 compute the tasks that would become enabled if the set of tasks $tc$ finishes. Variable $readyTaskSet$ initialized to $\emptyset$ at line 21 is used to keep these tasks. Function $traversed(s)$ represents the set of states (in the drawn automaton so far) that are traversed by some path from state $I$ to state $s$ ($I$ and $s$included). The union of the task sets associated with $traversed(s)$ denoted by $traversed(s).taskSet$ thus represents the tasks that have been executed before reaching $s$ and are executing in current state $s$. At line 22, the algorithm explores the tasks that have not been requested (denoted by $T - traversed(s).taskSet$) to find out $readyTaskSet$ once $tc$ finishes. Lines 23 to 25 decides whether $t_i$ is enabled once $tc$ finishes and adds $t_i$ to $readyTaskSet$ if it is. $t_i$ is enabled if the set of its precedent tasks is a subset of the union of tasks that have finished (denoted by $traversed(s) - s.taskSet$) and the tasks would finish denoted by $tc$. At line 27, *nextState* denotes the state following $s$ due to the tasks in $tc$ simultaneously finish. Its *taskSet* thus equals to the union of computed $readyTaskSet$ and the tasks that are still executing in $s$ after $tc$ finishes denoted by $s.taskSet - tc$. If $nextState.taskSet$ is $\emptyset$, this means that once

**Algorithm 1** Scheduler Automaton Derivation

```
 1: drawState(I);
 2: drawnStates = {I};
 3: stateQueue = stateQueue.add(I);
 4: for all t_i ∈ T do
 5:   if t_i.prec = ∅ then
 6:     readyTaskSet = readyTaskSet ⋃ t_i;
 7:   end if
 8: end for
 9: while stateQueue! = ∅ do
10:   s = stateQueue.popup();
11:   if s = I then
12:     nextState.taskSet = readyTaskSet;
13:     drawState(nextState);
14:     drawTrans(I, nextState, req/r_readyTaskSet);
15:     drawnStates = drawnStates ⋃ nextState;
16:     stateQueue.add(nextState);
17:   else if s = T then
18:     continue;
19:   else
20:     for all tc ∈ powerSet(s.taskSet) do
21:       readyTaskSet = ∅;
22:       for all t_i ∈ T− traversed(s).taskSet do
23:         if t_i.prec ⊆ (traversed(s).taskSet - s.taskSet) ⋃tc then
24:           readyTaskSet = readyTaskSet ⋃ t_i;
25:         end if
26:       end for
27:       nextState.taskSet = readyTaskSet ⋃ (s.taskSet - tc);
28:       if nextState.taskSet = ∅ then
29:         nextState = T;
30:       end if
31:       if  nextState ∈ drawnStates then
32:         drawTrans(s, nextState, e_tc);
33:       else
34:         drawState(nextState);
35:         drawTrans(s, nextState, e_tc);
36:         drawnStates = drawnStates ⋃ nextState;
37:         stateQueue.add(nextState);
38:       end if
39:     end for
40:   end if
41: end while
```

the tasks in *tc* finish, no more task can become enabled or is still executing, i.e., all tasks have finished executions and *nextState* is the end state $T$ (lines 28 to 30). In a scheduler automaton, a state might have more than one precedent states. The algorithm thus checks, at line 31, if *nextState* has been drawn. If it has, only the transition needs to be drawn from $s$ to *nextState* with label $\{e_{t_i} | t_i \in tc$ denoted by $e_{tc}$ at line 32. Otherwise, the algorithm draws both state

*nextState* and the transition at lines 34 and 35, and then updates *drawnStates* and *stateQueue* accordingly.

**Task execution behaviour.** In consideration of the four control points of task executions (see Section 3.1), the execution behaviour of task $A$ associated with two implementations (see Section 3.1) can be modelled as Figure 10. It features an initial *idle* state $I_A$, a *wait* state $W_A$, and two *executing* states $X_A^1$, $X_A^2$ corresponding to two implementations of task $A$. Controllable variables are integrated in the model to encode the controllable points: being delayed and executed. Upon the receipt of *start* request $r_A$, task $A$ goes to either:

- *executing* state $X_A^i, i \in \{1, 2\}$ if the value of *controllable* variable $c_i$ leading to $X_A^i$ is *true*, or
- *wait* state $W_A$ if delayed, i.e., the value of Boolean expression $c = \bigvee c_i, i \in \{1, 2\}$ is false.

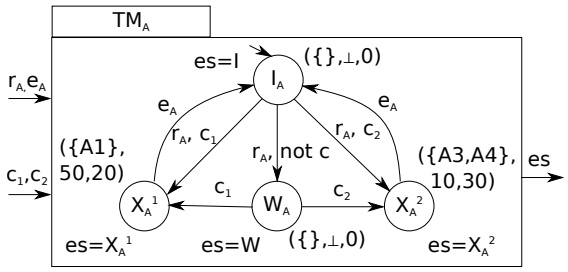



Figure 10: Execution behaviour model $TM_A$ of task $A$.

From wait state $W_A$, upon the receipt of event $c_i$, it goes to execution state $X_A^i$. When the execution of task $A$ finishes, i.e., the end notification event $e_A$ is received, the automaton goes back to *idle* state $I_A$. Output *es* represents its execution state.

*Local execution costs.* The execution costs of different task implementations are different. Three cost parameters are considered (see Section 3.1). We capture them by associating cost values denoted by a tuple $(rs, wt, pp)$ with the states of task models, where: $rs \in 2^{RA}$ ($RA$ is the set of architecture resources), $wt \in \mathbb{N}$ (a WCET value) and $pp \in \mathbb{N}$ (a power peak). The costs associated with *executing* states are the values associated with their corresponding implementations. For *idle* and *wait* states, apparently $rs = \emptyset, pp = 0$. However, the $wt$ values for *idle* and *wait* states depend on the execution times of their precedent tasks. We therefore represent it by using a special symbol $\perp$, and thus we have $wt \in \mathbb{N} \cup \perp$. Figure 10 gives the complete local model of task $A$.

**Global system behaviour model.** The parallel composition of control models for reconfigurable tiles $RM_1$-$RM_4$, battery $BM$ and tasks $TM_A$-$TM_D$, plus scheduler $Sdl$ comprises the system model:

$$\mathcal{S} = RM_1|...|RM_4|BM|TM_A|...|TM_D|Sdl$$

with initial state $q_0 = (Sle_1, ..., Sle_4, H, I_A, ..., I_D, I)$. It represents all the possible system execution behaviours in the absence of control (i.e., a runtime manager is yet integrated). Each execution behaviour corresponds to a *complete path*, which starts from initial state $q_0$ and reaches one of the *final states*:

$$Q_f = (q(RM_1), ..., q(RM_4), q(BM), I_A, ..., I_D, T),$$

where $q(Id)$ denotes an arbitrary state of automaton $Id$.

*Global costs.* The costs defined locally in each task execution model need to be combined into global costs.

*Costs on states.* A system state $q$ is a composition of local states (denoted by $q_1, ..., q_n$), and we define its cost from the local ones as follows:

- used resources: union of used resources associated with the local states, i.e., $rs(q) = \bigcup rs(q_i), 1 \leq i \leq n$;
- worst case execution time: this indicates how much time the system takes at most in this current state. It is thus defined as the minimal WCET of all executing tasks in this state, i.e., $wt(q) = min(wt(q_i), wt(q_i) \neq \perp, 1 \leq i \leq n)$; Otherwise, if no task is executing in the state, i.e., $\forall 1 \leq i \leq n, wt(q_i) = \perp$, $wt(q) = 0$;
- power peak: the sum of values associated with the local states, i.e., $pp(q) = \sum(pp(q_i), 1 \leq i \leq n)$;
- worst case energy consumption: the product of the worst case execution time and power peak of the system state, i.e., $we(q) = pp(q) * wt(q)$.

*Costs on paths.* We also need to define the costs associated with paths so as to capture the characteristics of system execution behaviours. Given path $p = q_i \rightarrow q_{i+1} \rightarrow ... \rightarrow q_{i+k}$, and costs associated with system states, we define corresponding costs on path $p$ as follows:

- WCET: the sum of WCETs on the states along the path, i.e., $wt(p) = \sum wt(q_j), i \leq j \leq i + k$;
- power peak: the maximum value on the states along the path, i.e., $pp(p) = max(pp(q_j), i \leq j \leq i + k)$;
- worst case energy consumption: the sum of the worst case energy consumptions on the states along the path, i.e., $we(p) = \sum we(q_j), i \leq j \leq i + k$.

**System objectives.** The two types of system objectives: logical and optimal ones, can then be described in terms of the states and the costs defined on the states or paths of the model.

*Logical control objectives.* For any system state $q$, we want to enforce the following:

(1) exclusive uses of reconfigurable tiles by tasks: $\forall q_i, q_j \in q, i \neq j$, that $rs(q_i) \bigcap rs(q_j) = \emptyset$;

(2) dual accesses to shared memory, i.e., at most two functions can access the memory at the same time:
$\sum v_i \leq 2$, s.t. $v_i = \begin{cases} 1 & q_i \in X_i \\ 0 & \text{otherwise} \end{cases}$, where $X_i$ represents the set of executing states of corresponding task;

(3.a) switch tile $A_i$ to sleep mode, when executing no task: $\nexists q_i \in q, Ai \in rs(q_i) \Rightarrow act_i = false$;

(3.b) switch tile $A_i$ to active mode when executing task(s): $\exists q_i \in q, Ai \in rs(q_i) \Rightarrow act_i = true$;

(4) reachability: $Q_f$ is always reachable.

(5) battery-level constrained power peak (given threshold values $P_0, P_1, P_2$): $pp(q) < P_0$ (resp. $P_1$ and $P_2$) when battery level is high (resp. medium and low).

*Optimal control objectives.* Such objectives can be further classified into two types of objectives: one-step optimal and optimal control on path objectives. We use pseudo functions *max* and *min* in the following to represent the maximisation and minimisation objectives, respectively.

*One-step optimal objectives.* One-step optimal objectives aim to minimise or maximise costs associated with states and/or transitions in a single step [12]. Objective 4 of Section 3.1 belongs to this type.

(6) minimise power peak *pp* in next states of state $q$: $min(pp, q)$.

*Optimal control on path objectives.* Such objectives aim to drive the system from the current state to the target states $Q_f$ at the best cost [5]. Objective 5 and 6 are such objectives.

(7) minimise remaining WCET *wt* from state $q$: $min(wt, q, Q_f)$;

(8) minimise remaining energy consumption *we* from $q$: $min(we, q, Q_f)$.

## 3.3 BZR encoding and DCS

Given the system graphical models and objectives of Section 3.2, this section describes the controller synthesis by using BZR and the DCS tool Sigali. Logical and optimal objectives are treated differently.

**BZR encoding of the system model.** The automaton encoding of system components in Section 3.2 can be translated to textual encoding easily as Figure 2. The BZR encoding of the global system behaviour can then be obtained by composing all these models. Finally, the costs on system states are defined as described in Section 3.2 (not detailed here, due to space limitation).

**Enforcing logical control objectives.** BZR *contracts* are able to directly encode the logical control objectives of Section 3.2. The following shows the BZR *contract*.

```
1  contract
2  var exclusive_tileA1, idle_tileA1, swt_sleep_tileA1,
      swt_act_tileA1, bound_pp: bool;
3  let
     (*exclusive usage of tile A1*)
4    exclusive_tileA1 = idle_tileA1 or only_A_on_tileA1
       or ... or only_D_on_tileA1;
5    idle_tileA1 = not A_on_tileA1 & not B_on_tileA1
       & not C_on_tileA1 & not D_on_tileA1;
6    only_A_on_tileA1 = A_on_tileA1 & not B_on_tileA1
       & not C_on_tileA1 & not D_on_tileA1;
     (*switch to sleep mode when running no task*)
7     swt_sleep_tileA1 = not idle_tileA1 or not act1;
     (*switch to active mode when executing a task*)
8     swt_act_tileA1 = idle_tileA1 or act1;
     (*bounded power peak*)
9     bound_pp = if battery_high then (pp <= 500)
      else if battery_low then (pp <= 300)
      else (pp <= 400);
10 tel
11 enforce exclusive_tileA1 & swt_sleep_tileA1
     & swt_act_tileA1 & bound_pp
12 with (c_a1, c_a2, c_a3, c_a4, c_1,c_2, ... : bool)
```

Line 2 declares the local variables used within the contract with keyword *var*. They are declared as boolean variables by using keyword *bool*, and are defined in the body, i.e., Lines 4 to 9 between keywords *let* and *tel*. Variable $exclusive\_tileA1$ represents the exclusive usage of tile $A1$ (objective 1), i.e., objective 1, and is defined as the disjunction of five possible cases: tile $A1$ is idle denoted by $idle\_A1$; only task $T$ is running on tile $A1$ denoted by $only\_T\_on\_tileA1, T = \{A, B, C, D\}$. $idle\_A1$ and $only\_T\_on\_tileA1$ are defined at Lines 5 and 6 based on the states of the four task implementation models. $T\_on\_tileA1$ represents that task $T$ is using tile $A1$. objectives 2.*a* and 2.*b* Objectives 2.*a* and 2.*b* denoted by $swt\_sleep\_tileA1$ and $swt\_act\_tileA1$ are defined at Lines 7 and 8 by using the equivalent expressions of $idle\_A1 \Rightarrow not\ act1$ and $not\ idle\_A1 \Rightarrow act1$ respectively. $act1$ is the output of tile $A1$'s behavior model (see Figure 8). Line 9 defines the objective 3 denoted by $bound\_pp$. These objectives are then enforced at Line 11 with keyword $enforce$. All the relevant controllable variables to be computed are declared at Line 12 with keyword *with*. They are the controllable variables declared in the tile models (see Figure 8) and task implementation models (see Figure 10). The exclusive usages of the other tiles and their mode switch managements can be encoded similarly.

Taking as input the system model and the contract, the BZR compiler can synthesise a controller (in C or Java code) automatically satisfying the defined objectives. There is also a graphical tool enabling the users to perform simulations of the controlled system by combines the controller with the system model.

**Enforcing optimal control objectives.** Optimal control objectives can be addressed by combining BZR and Sigali. In this case, the BZR compiler serves as the front end to encode system behaviour, and produces an intermediate file. The optimal control objectives can then be encoded and integrated into this file, which feeds the Sigali tool. The DCS is finally performed by Sigali to automatically generate a controller satisfying the objectives. The generated controller can also be combined with the system model by the BZR compiler for simulations.

# 4 Experimental results

## 4.1 Experimental validation

In this section we describe an experimental case study and demonstrator, where some of the previous control models and objectives are applied to a concrete FPGA based platform.

### 4.1.1 Case study

We consider a video processing system to be implemented on a platform containing an FPGA, so that the partial reconfigurations of the FPGA controlled by a synthesized controlled can be visualised. The processing system consists of a camera capturing images, a dispatcher feeding image pixels to the platform, a compositor aggregating pixels produced by the FPGA, and a screen displaying the processed images. Each captured image is divided into 9 areas, and the processing of each area is taken care by one dedicated reconfigurable tile of the FPGA. The FPGA is thus divided into nine tiles (the same way we had 4 in Figure 5). In this way, when a tile is reconfigured, one can see it on the screen. Three possible filtering algorithms (namely red, green and blue ones) are considered to be implemented on each reconfigurable tile to process images. When configured to process the same image, they have different performance values regarding some characteristics such as power peak, execution time. In the study, we suppose the power peaks of each tile for running the three filters are 3, 2 and 1.

The processing system can work at two different modes: *high* and *low* controlled by the user through a switch on the platform. The user can also demand the use of red filters for the processing of the four corner areas of images by using another switch. Apart from the user demands, the system should also respect the following three rules: 1) four *corner areas* of images are of the same colour, and the rest areas are of the same colour; 2) the global power peaks of the platform are bounded by 30 (and 20) in *high* (and *low* resp.) mode; 3) maximising the power peaks of next states. A runtime manager is thus required to configure the nine tiles to filter images in the way satisfying the aforementioned requirements.

### 4.1.2 Controller integration

Following the design flow in Section 3.2, the C code of the runtime manager was generated within 5 seconds by BZR. The manager then needs to be integrated with the system. This section describes the structure of the generated code and the way to combine it with the system.

The manager code is composed of two C code folders with overall size 77.8 kilobytes.One folder contains the C code of the generated controller which computes the values of controllable variables according to system states and inputs; the other folder contains the code for keeping track of the system states by performing state transitions according to system inputs, states and the values of computed controllable variables (this is done by invoking the functions in the former folder. Two additional C files named `main.c` and `main.h` are also generated by the compiler for simulation purpose. However, they can be easily adapted to serve as the interface between the manager and the system. Next we take a closer look at the `main.c` code, and then describe the way of adapting them such that the manager code can be combined with the system.

The code of `main.c` can be divided into three parts: *i*) input part: system input variables declared in the system model; *ii*) system model declaration and initialisation part: state variables and output variables (named as `mem` and `res` respectively) declaration and system state initialisation (by `reset(&mem)`); *iii*) system states tracking and transition part: it is an infinite loop, and each iteration consists of the input variable evaluation, a step function: `sys_step(`*inputs*`, &mem)` computing output `res` and updating system state `&mem` according to the inputs and current state `&mem`, and the printout of output variable values. To integrate the manager with the system, one needs to 1) pass the system input values to the input variables of the manager; 2) define within the infinite loop the timing to invoke the step function; and 3) interpret the output variables as system (reconfiguration) actions. In the next section, we show how the generated manager can be integrated with a Xilinx platform.

### 4.1.3 System implementation

We have implemented the video processing system on an ML605 board from Xilinx. It includes a Virtex-6 FPGA (XC6VLX240T), several I/O interfaces like switches, buttons, Compact Flash reader, and an external 512MB DDR3 memory. An Avnet extension card (DVI I/O FMC Module) with 2 HDMI connectors (In and Out) has been plugged onto the platform so that it can receive and send video streams through the connectors.

Figure 11 illustrates the global structure of our implementation. We have divided the FPGA surface into two regions: static and reconfigurable regions. Nine independent reconfigurable tiles are specified in the reconfigurable region. The tiles are in charge of the video processing tasks described in Section 4.1.1. The microblaze is a 32-bit soft-core processor synthesised on the static region of the FPGA (like *A*0 in Figure 4). It executes two main system tasks: the computed manager and the management of the configuration bitstreams. The latter task involves the control of related peripherals (i.e., Compact Flash memory, I/O interrupts, DDR3, ICAP) through corresponding implemented controllers. The external DDR 3 memory is used to buffer

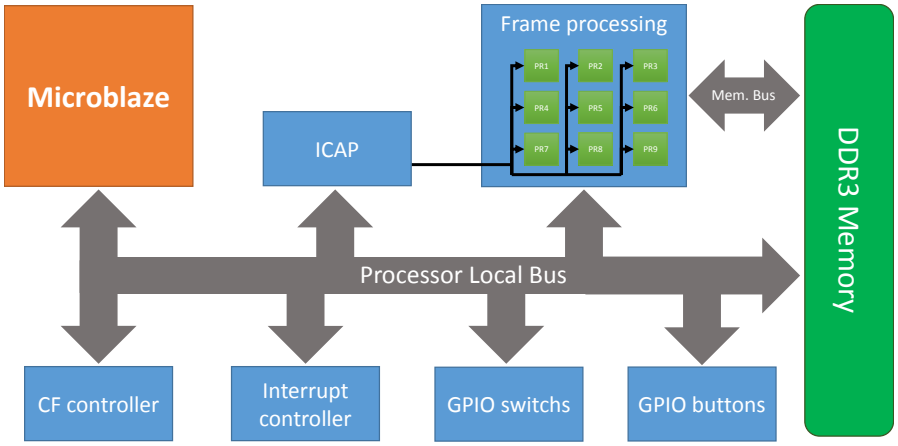



Figure 11: Global structure of the implementation

the frame pixel data of video streams, and store the software executable to be launched by the microblaze. The accesses to the DDR by the FPGA are managed by the DDR controller. We use a compact Flash card to store the bitstreams of the filter implementations on each reconfigurable tile. The C code of the manager is deployed on the microblaze as an infinite loop. It is invoked whenever the microblaze is interrupted. Two additional interrupt controllers (GPIO switches and GPIO buttons) are added for the platform to generate interrupts. They monitor the states of the buttons and switches, and generate interrupts when these states change. Once the manager is invoked, it is able to read the states of the switches and computes out a new configuration. The microblaze then selects the appropriate bitstreams from the Compact Card, and sends them to the ICAP to reconfigure the nice reconfigurable tiles.

Experimental results show that the integrated manager did meet the system requirements mentioned in Section 4.1.1 after a considerable number of tests.

## 4.2 Scalability

We have carry on extensive experiments to evaluate the scalability of our framework. Table 1 shows our experimental results to compute the *controller* by using the tools. It gives the time costs for different DCS operations corresponding to different system objectives w.r.t. different system models and state space sizes. The state space size of each system model is computed by simply multiplying state space sizes of its composed automata. The size of synthesized controllers varies from 50Kb (objective 2 on model 4:(2,3,2,3)) to 28Mb (objective 7 on model 6:($1^6$)). We have started our experiments from the task graph of Figure 6. We then refine B to 3 tasks so as to increase the system model to 6 tasks, and at last, refine C to 3 tasks as well to address a 8 task model. We use the notation $n : (m_1, ...m_n)$ to represent the models, where $n$ denotes the number of tasks, and $m_i$ the number of possible implementations of task $i$. Besides, we use $m^k$ to represents $k$ consecutive $m$'s. E.g., $4 : (4^4)$ denotes $4 : (4, 4, 4, 4)$. All experiments are performed on a computer with a Intel(R) Core(TM)2 Duo CPU of 2.33GHz and a 3.8Gb main memory. Due to space limitation, details about encoding of our models in BZR and Sigali are left out.

In our experiments, the DCS of invariance constraints, i.e., objectives 1-3,5, are applied directly on the original system model. On the basis of the resulting controller, the optimal and reachability ones are then performed. The objectives about invariance and reachability appear

promising, while optimal ones are, unsurprisingly, explosive. An interesting point observed is that the time cost is not always increasing as state space size grows. System models consisting of more functions but less possible implementations could have less synthesis times, e.g., DCS operations for 6:($1^6$) model take less times than these for 4:($4^4$) model. The reason may come from the fact that one-step-optimal and optimal-on-path syntheses require to examine the costs of next states, while more implementations of functions mean more choices to explore. Due to time and resource limitations, we have decided to stop the synthesis processes if not finished after 3 days of computation.

| system model & **state space size** / target objectives | 4:(2,3,2,3) **241,920** | 4:($4^4$) **806,736** | 6:($1^6$) **5,898,240** | 6:(3,1,1,2,1,3) **16,588,800** | 6:(3,$2^4$,3) **32,400,000** | 8:($1^8$) **566,231,040** | 8($3^3$,$2^5$) **5,832,000,000** |
|---|---|---|---|---|---|---|---|
| 1) exclusive usage of $A_1$-$A_4$ | 0.29sec | 0.65sec | 0.16sec | 1.16sec | 8.20sec | 1.10sec | 16min1sec |
| 2) dual access to memory | 0.12sec | 0.49sec | 0.10sec | 0.69sec | 1.50sec | 1.29sec | 23.05sec |
| 3.a) switch to active mode | 0.74sec | 2.28sec | 0.46sec | 1.88sec | 1min19sec | 1.30sec | 27min58sec |
| 3.b) switch to sleep mode | 0.76sec | 2.01sec | 0.22sec | 1.74sec | 2min11sec | 0.90sec | 41min29sec |
| 5) battery-level constrained p.p. | 0.89sec | 2.23sec | 0.68sec | 4.18sec | 21.18sec | 2.21sec | 49min24sec |
| 4) reachability: | 1.78sec | 3.48sec | 4.74sec | 17.33sec | 2min | 25.16sec | 3hr34min |
| 6) minimize p.p. in next states | 3min54sec | 3hr18min | 29min45sec | *stopped* | *stopped* | *stopped* | *stopped* |
| 7)* minimize remaining WCET: | 9min17sec | 2hr43min | 21min19sec | *stopped* | *stopped* | *stopped* | *stopped* |

Table 1: The time costs for DCS operations corresponding to different target objectives w.r.t. different system models and state space sizes. $n : (m_1, ...m_n)$ denotes a model of $n$ functions, with $m_i$ denoting the number of possible implementations of function $Fi$. $*$ Objectives 7 and 8 are the same kind of operation (objective 8 is thus omitted here).

# 5 Related work

Computing systems based on reconfigurable architectures can draw various benefits, such as adaptability and efficient acceleration of compute-intensive tasks [14]. However, the dynamic reconfiguration capabilities of such architectures further complicate their design, and require more efforts to maintain such complex infrastructure. Adopting self-adaptive and autonomic computing systems is one solution to address these problems. Companies like IBM and Intel have invested in this research.

Generally speaking, existing decision-making techniques for self-optimisation of autonomic systems can be classified into three categories: heuristics, control-theory, and machine learning [10]. Control techniques are able to provide formal performance guarantees compared to the other two techniques. Existing approaches applying standard control techniques such as Proportional Integral and Derivative controllers or Petri nets are discussed [10]. However, discrete control has only been seldom applied [7], and to the best of our knowledge, only few works have targeted at computing systems on reconfigurable architectures.

In [16], a reconfigurable architecture based evolvable system exploiting self-adaptive techniques is proposed. It is one of the first implementations of a FPGA-based self-aware adaptive system. It adopts the application heartbeats as monitoring framework, and a heuristic mechanism to switch between configurations. Self-management in the form of self-healing exploiting FPGAs is proposed in [9]. However, the approach does not involve control. A new architectural proposal in [11] provides a slot-based organisation of the reconfigurable hardware and an elaborate communication framework with good reconfiguration support. The focus is, however, on infrastructure aspects rather than on control.

Our approach is closer to that in [6], but we focus on logical aspects and discrete control. In [13], a design flow, from high level models to automatic code generation, for the implementation of reconfigurable FPGA based SoCs is proposed. The system control aspects need to be modeled manually and integrated into the flow, while we advocate automatic controller synthesis. Compared to [7], we have applied more elaborate DCS algorithms, and the integration into a design flow and compilation chain is more developed.

# 6 Conclusion and Perspectives

Reconfigurable architectures, especially DPR FPGAs, constitute a platform for adaptive computing that is gaining widespread use. They are a typical target for autonomic computing approaches, although they are not often explicitly tackled that way. The contribution of the paper is manifold: *i*) we propose a systematic modelling framework for DPR FPGA based embedded systems, where application behaviour (defined by a task graph), task implementations and executions (characterised by parameters of interest e.g., time and power consumptions), architecture resource uses and reconfigurations are modelled separately by using automata; *ii*) we apply formalisms and tools from discrete control, supported by a programming language and synthesis tool, to encode and perform the computation of an autonomic manager as a DCS problem; *iii*) we perform an experimental validation of our proposal by implementing a video processing system on a Xilinx FPGA platform.

Perspectives include the ongoing work to enrich our models with reconfiguration costs, e.g., memory access for bitstreams on local or secondary storage, and communication aspects. Another interesting direction is to exploit the modular synthesis and compilation [4] for manager computing. Except for its specification and modelling structuring benefits, it also provides a means to address the scalability issue by decomposing big problems in a way that breaks down the combinatorial complexity.

# References

[1] Karine Altisen, Aurélie Clodic, Florence Maraninchi, and Éric Rutten. Using controller-synthesis techniques to build property-enforcing layers. In *Proceedings of the European Symposium on Programming (ESOP'03)*, pages 174–188, 2003.

[2] Xin An, Eric Rutten, Jean-Philippe Diguet, Nicolas le Griguer, and Abdoulaye Gamatié. Autonomic management of dynamically partially reconfigurable fpga architectures using discrete control. In *Proc. of the 10th International Conference on Autonomic Computing (ICAC'13)*, June 2013.

[3] C. Cassandras and S. Lafortune. *Introduction to Discrete Event Systems*. Kluwer Academic Publishers, 2008.

[4] G. Delaval, H. Marchand, and E. Rutten. Contracts for modular discrete controller synthesis. In *Conf. on Languages, Compilers, and Tools for Embedded Systems*, pages 57–66, 2010.

[5] E. Dumitrescu, A. Girault, H. Marchand, and E. Rutten. Multicriteria optimal discrete controller synthesis for fault-tolerant tasks. In *Workshop on Discrete Event Systems*, pages 356–363, 2010.

[6] Y. Eustache and J.-P. Diguet. Specification and os-based implementation of self-adaptive, hardware/software embedded systems. In *Conf. on Hardware/Software codesign and system synthesis (CODES/ISSS)*, pages 67–72, 2008.

[7] Sébastien Guillet, Florent de Lamotte, Nicolas Le Griguer, Eric Rutten, Guy Gogniat, and Jean-Philippe Diguet. Designing formal reconfiguration control using uml/marte. In *Reconfigurable and Communication Centric Systems-on-Chip, ReCoSoC*, 2012.

[8] J. Hellerstein, Y. Diao, S. Parekh, and D. Tilbury. *Feedback Control of Computing Systems*. Wiley, 2004.

[9] S. Jovanovic, C. Tanougast, and S. Weber. A new self-managing hardware design approach for fpga-based reconfigurable systems. In *Reconfigurable Computing: Architectures, Tools and Applications*, volume 4943 of *LNCS*. 2008.

[10] Martina Maggio, Henry Hoffmann, Alessandro V. Papadopoulos, Jacopo Panerati, Marco D. Santambrogio, Anant Agarwal, and Alberto Leva. Comparison of decision-making strategies for self-optimization in autonomic computing systems. *ACM Trans. Auton. Adapt. Syst.*, 7(4):36:1–36:32, 2012.

[11] Mateusz Majer, Jürgen Teich, Ali Ahmadinia, and Christophe Bobda. The erlangen slot machine: A dynamically reconfigurable fpga-based computer. *The Journal of VLSI Signal Processing Systems for Signal, Image, and Video Technology*, 47:15–31, 2007.

[12] H. Marchand and M. Samaan. Incremental design of a power transformer station controller using a controller synthesis methodology. *IEEE Trans. on Soft. Eng.*, 26(8):729 –741, 2000.

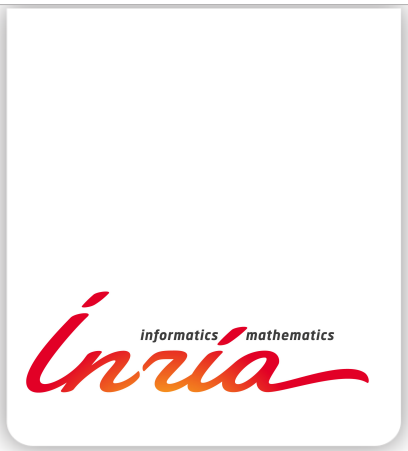
[13] Imran Rafiq Quadri, Huafeng Yu, Abdoulaye Gamatié, Eric Rutten, Samy Meftali, and Jean-Luc Dekeyser. Targeting reconfigurable fpga based socs using the uml marte profile: from high abstraction levels to code generation. *IJES*, 4(3/4):204–224, 2010.

[14] Marco D. Santambrogio. From reconfigurable architectures to self-adaptive autonomic systems. *IJES*, 4(3/4):172–181, 2010.

[15] S. Sicard, F. Boyer, and N. De Palma. Using components for architecture-based management: the self-repair case. In *Proc. Conf. ICSE*, 2008.

[16] F. Sironi, M. Triverio, H. Hoffmann, M. Maggio, and M.D. Santambrogio. Self-aware adaptation in FPGA-based systems. In *Field Programmable Logic and Applications*, 2010.

[17] Y. Wang, S. Lafortune, T. Kelly, M. Kudlur, and S. Mahlke. The Theory of Deadlock Avoidance via Discrete Control. In *Conf. POPL*, 2009.




**RESEARCH CENTRE**
**GRENOBLE – RHÔNE-ALPES**

Inovallée
655 avenue de l'Europe Montbonnot
38334 Saint Ismier Cedex

Publisher
Inria
Domaine de Voluceau - Rocquencourt
BP 105 - 78153 Le Chesnay Cedex
inria.fr

ISSN 0249-6399